1. 上课约定须知

课程主题：SparkCore 第四次课：SparkCore 源码分析 -- Task执行 和 SparkShuffle 源码剖析
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次作业复盘

使用 Spark 的 RPC 框架实现一款 C/S 架构的聊天应用程序。

要求：

```
1、该应用应用程序，有最基本的服务端程序和客户端程序
2、首先启动服务端，保证服务端一直运行
3、然后启动客户端，客户端向服务端注册
4、再启动其他多个客户端，待启动的客户端注册之后，和其他客户端（当然，当前客户端必须具备感知其他客户端存在的功能，然后选择通信对象）进行通信。
5、客户端关闭，其他客户端收到通知。
```

3. 上次内容总结

上次课程讲解的主要内容是：Spark Application 提交 和 SparkContext 的初始化源码分析

```
1、Spark Application 应用程序编写套路总结，找到程序执行入口
2、spark-submit 脚本分析
3、SparkSubmit 类分析
4、Client提交Job到Master过程解析
5、Driver启动和SparkContext初始化源码详解
    SparkConf 初始化
    SparkEnv 初始化
    DADScheduler，TaskScheduler，SchedulerBackEnd 初始化
    SchedulerBackEnd启动
6、Applicatoion注册
7、Executor启动分析
```

再次，顺便复习一下 Spark Job 的提交执行完整流程：

```
01、用户根据 Spark 提供的应用程序编写套路，编写用户处理逻辑的应用程序，然后打成 jar 包
02、用户通过 Spark 的 spark-submit 脚本来提交 spark job
03、执行 SparkSubmit 类的 main() 方法
04、如果提交运行的资源系统对象为 Spark Standalone 集群，则 SparkSubmit 内部通过 ClientApp 来执行提交
05、在 ClientApp 内部，会初始化一个 CLientEndponit 组件，发送 RequestSubmitDriver 消息给 Master
06、Master 处理 RequestSubmitDriver 消息，创建 DriverInfo 并执行 Driver 注册，然后给 CLientEndponit 返回 SubmitDriverResponse 消息，同时 Master 也会调用 schedule() 来启动 Driver 和 Executor
07、Master 发送 LaunchDriver 给某一个 Worker 来启动 Driver
08、Worker 接收到 LaunchDriver 消息然后启动：DriverWrapper，然后跳转到 DriverDrapper 的 main() 方法来启动一个新的 JVM 进程
09、在 DriverDrapper 中会通过反射的方式来执行 用户自定义应用程序的 main() 方法
10、执行 SparkSession(内部就是初始化 SparkConf SparkContext) 的初始化
11、初始化 SparkContext：TaskScheduler SchedulerBackend（DriverEndpoint CLientEndpoint） DAGScheduler
12、启动 SchedulerBackend，内部会通过 ClientEndpoint 发送 RegisterApplication 消息给 Master 进行应用注册，接收 Master 返回的 RegisteredApplication 消息。
13、Master 再次调用 schedule() 方法来通过发送 LaunchExecutor 给 Worker 调度 Executor 的运行， Executor 进程的真正启动类是：ExecutorBackend
14、ExecutorBackend 启动的时候，其实内部初始化了一个 Exeuctor，Exeuctor 的内部初始化了一个线程池，这个线程池就是等待 Driver 发送 Task 过来执行的
15、Worker 启动好了 Executor 之后会给 Master 发送反馈消息 ExecutorAdded，会发送 RegisterExecutor 注册消息给 Driver
16、当 Driver 接收到 所有 Executor 的注册消息的时候，意味着，该启动的所有运行 Task 的 jvm 进程全部启动完毕，此时 SparkContext 初始化搞定了。
```

接下来就是 submit job 然后 run 起来！

4. 本次内容大纲

在上一次课程讲完 Driver 和 Exeuctor 的启动之后，今天讲解的内容是 Spark Application 的提交执行和 具体的 Task 执行细节的源码剖析。主要内容项包含如下几点：

```
1、Spark Job 提交分析
2、RDD 的 DAG 构建和 Stage 切分源码详解
3、TaskScheduler 提交 Task 分析
4、Spark Task 执行源码详解
5、Spark shuffle 源码详解
```

5. 详细课堂内容

5.1. Spark Job 提交分析

Spark 应用程序的编写套路：

```
1、获取程序执行入口：SparkSession
2、通过 SparkSession 加载数据源得到数据抽象对象 DataSet
3、针对 DataSet 调用各种逻辑操作算子来执行业务处理
4、调用 action 算子来触发 job 的提交
5、关闭 SparkSession
```

在上一次课程中，主要讲解到的是第一步（初始化 SparkSession）过程中的一些细节，主要是 Driver 和 Executor 的启动。今天重点来讲解第四步（当调用action 算子的时候，就触发了一个 job 的提交执行）。

所以在用户自定义的业务程序代码中，当执行到 action 算子的时候，就真正提交了一个 job 到 Spark 集群去运行。

在 action 算子的内部，都是通过如下代码进行提交的：

```
sc.runJob(this, ....)
```

注意：

```
sc: SparkContext
this：当前job的最后一个RDD
```

SparkContext 之后，转交给 DAGScheduler 来执行提交！DAGScheduler 的 submitJob(...) 方法，其实是封装了一个 JobSubmitted 事件，然后提交给 DAGScheduler 初始化的时候启动的 DAGSchedulerEventProcessLoop，其实是加入到 eventQueue 队列中。然后 DAGScheduler 内部的 eventThread 一直阻塞在 eventQueue.take() 上获取可消费的事件。当提交一个 JobSubmitted 事件的时候，回调 DAGSchedulerEventProcessLoop 的 onReceive() 方法，其实是触发了 DAGScheduler 的 doOnReceive(event) 方法执行事件的处理。

5.2. RDD 的 DAG 构建和 Stage 切分源码详解

通过上述的分析，可以得知，最终代码跳转到 DAGScheduler 的 handleJobSubmitted(....) 方法来执行处理。请看消息信息：

```
// DAGScheduler 提交 job 其实是构建了一个 JobSubmitted 事件给
DAGSchedulerEventProcessLoop
DAGSchedulerEventProcessLoop.onReceive(event: DAGSchedulerEvent)

    // 根据 JobSubmitted 调用 DAGScheduler 的 handleJobSubmitted() 方法执行处理
    dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite,
listener, properties)

        // 第一件重要的事情：stage 切分
        finalStage = createResultStage(finalRDD, func, partitions, jobId,
callSite)

        // 第二件重要的事情：提交 stage 执行
        submitStage(finalStage)
```

为了能看懂源码，先了解一些基础知识：

```
1、一个 Spark Job 一般来说，肯定是分成多阶段（stage）并行执行（Task）
2、最后一个阶段，称之为： ResultStage，除了最后一个阶段之外的前面所有阶段，都叫做
ShuffleMapStage
3、不管是 ResultStage 还是 ShuffleMapStage，都会并行执行多个 Task
4、ResultStage 中执行的 Task 叫做 ResultTask，ShuffleMapStage 当中执行的 Task 叫做
ShuffleMapTask
```

举例：

```
如果一个 job 有 3 个 shuffle， 那么就会被切分成 4 个 Stage，前面三个叫做：
ShuffleMapStage， 最后一个叫做 ResultStage
一定是 第一个阶段执行完毕之后，才可以开始执行第二个阶段。
每个阶段，执行的时候，都是并发成多个 Task 同时运行！
```

Spark 切分 Stage 的思路：

```
因为算子执行的时候，就已经构建了 DAG，在做 stage 切分的时候，就从 DAG 的后面往前面扫描，如果
遇到一个 shuffleDependency，就切断，并且形成为一个 Stage，然后放入到一个 栈 中： 最先生成
的 Stage是最后执行的 Stage，同时也是第一个被放入到 栈 中的 Stage，所以将来要执行 Stage 的时
候，是从栈顶 遍历到 栈底，最后的 Stage 的执行顺序也就自然变成了 从前stage 执行到 后面的
stage
```

举例：

```
RDD1 ---> RDD2 ---> RDD3  ---> RDD4  ---> RDD5  ---> RDD6
```

在上述代码中，其实可以看出来 DAGScheduler 做了一件最重要的事情就是： DAG 的 Stage 切分！

```
// 创建 ResultStage
1、createResultStage(传入finalRDD获得ResultStage)        ->2

    // 先找可能存在 ParentStage，如果有，也构建出来
    2、getOrCreateParentStages(传入rdd获得父stage)            ->3->4

        // 第一步：先找当前 rdd（finalRDD） 的所有 直接 父 shuffleDependency
        3、getShuffleDependencies(传入rdd获得宽依赖)
```

```
        // 第二步：把 直接 父 shuffleDependency 编程 ShuffleMapStage 对象
        4、getOrCreateShuffleMapStage(传入宽依赖获得ShuffleMapStage)
->5->6

            // 当前要构建的 ShuffleDependency 可能还要依赖于其他的 ShuffleDependency
            // 执行递归构建
            5、getMissingAncestorShuffleDependencies(传入一个rdd获得所有宽依赖)
->3

            // 把 ShuffleDependency 构建成 ShuffleMapStage
            6、createShuffleMapStage(传入宽依赖获得ShuffleMapStage)                ->2

                // 调用构造器来实现创建 ShuffleMapStage
                7、val stage = new ShuffleMapStage(....)

    // 最终，无论如何，一定会有至少一个stage，最后的一个stage 称之为： ResultStage
    8、finalStage = new ResultStage

// 提交 ResultStage
9、submitStage(finalStage)
```

将来提交 Stage 的时候，虽然提交的是：ResultStage。 但是这个 ResultStage 可能依赖于其他的多个父 ShuffleMapStage，所以执行执行递归的调度 S1 <----- S2 <------ S3 ，最终调用了 submitStage(finalStage) 提交了 ResultStage 的执行， 底层调用：taskScheduler.submitTasks(new TaskSet()) 之后，此时 DAGSchedueIr 就没用了。

核心入口：

```
// 提交 ResultStage 执行
DagScheduler.submitStage(finalStage);

    // 递归提交
    submitMissingTasks(stage, jobId.get)

        // 一个 Stage 变成一个 TaskSet 然后通过 TaskScheduler 执行提交
        taskScheduler.submitTasks(new TaskSet(tasks.toArray, stage.id,
stage.latestInfo.attemptNumber, jobId, properties))
```

到此，DAGScheduler 的工作完成。

5.3. TaskScheduler 提交 Task 分析

按照上述描述：TaskScheduler 提交 Task 的具体逻辑的入口：

```
taskScheduler.submitTasks(new TaskSet(tasks.toArray, stage.id,
stage.latestInfo.attemptNumber, jobId, properties));
```

内部调用：

```
backend.reviveOffers();
```

开始提交 Task：

```
// 开始执行 Task
launchTasks(taskDescs);

    // Driver 给 CoarseGrainedExecutorBackend 发送 LaunchTask 消息用来启动和执行一个
Task
    executorData.executorEndpoint.send(LaunchTask(new
SerializableBuffer(serializedTask)));
```

发送 LaunchTask 消息给 Executor。当 CoarseGrainedExecutorBackend 接收到 LaunchTask 消息，则开始执行调用对应的 Executor 来执行 Task：

```
executor.launchTask(this, taskDesc)
```

具体的内部实现是：

```
// 先封装一个 TaskRunner
val tr = new TaskRunner(context, taskDescription)
runningTasks.put(taskDescription.taskId, tr)

// 调用 Exeuctor 内部的线程池来执行 Task
threadPool.execute(tr)
```

此刻，终于把 Task 提交到 Worker 上的 Executor 中的线程池来进行运行了！

5.4. Spark Task 执行源码详解

Spark Task 的执行入口是：

```
Task.runTask(context);
```

对于 SparkCore 中的 Task 有两种：

```
1、如果提交的是 ShuffleMapStage ， 则运行的 Task 是 ShuffleMapTask
2、如果提交的是 ResultStage， 则运行的 Task 是 ResultTask
```

且看 ShuffleMapTask 的 runTask() 方法的内部核心代码：

```
// 通过反序列化得到 RDD 和 ShuffleDependency 的信息
val ser = SparkEnv.get.closureSerializer.newInstance()
val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])]
(ByteBuffer.wrap(taskBinary.value), ...)

// 获取 Shuffle 过程中的 Writer
var writer: ShuffleWriter[Any, Any] = null
val manager = SparkEnv.get.shuffleManager
writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)

// 执行 RDD 指定分区数据数据的逻辑计算，并且执行 shuffle
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <:
Product2[Any, Any]]])
```

```
// 删除 Task 执行过程中的 spill 文件，释放内存和磁盘资源
writer.stop(success = true).get
```

两个重点：

```
// 执行计算
rdd.iterator(partition, context)
    // 有缓存
    getOrCompute(split, context)
    // 没有缓存
    computeOrReadCheckpoint(split, context)

// 执行溢写
// 后面这一整段代码的逻辑是： SortShuffleWriter 中的逻辑
// SortShuffleManager其实有两种机制：
// 第一种运行机制： 普通运行机制： writer = SortShuffleWriter
// 第二种运行机制： bypass 机制： writer =
writer.write(records)
    // 获取一个排序器
    sorter = new ExternalSorter[K, V, C](.....)

    // records 其实是一个迭代器，通过迭代的方式，把所有数据，都写入到 sorter 中
    // 然后每次写入一条数据到 map或者buffer 中的时候，都会执行 maybeSpillCollection 判断
是否需要执行 spill 操作
    sorter.insertAll(records)

        // 在写入到 sorter 的时候，如果会进行局部聚合操作，则使用 map 数据结构来存储结果数据
        map.changeValue((getPartition(kv._1), kv._1), update)
        maybeSpillCollection(usingMap = true)

        // 在写入到 sorter 的时候，如果不进行局部聚合操作，则使用 buffer 数据结构来存储结果
数据
        buffer.insert(getPartition(kv._1), kv._1, kv._2.asInstanceOf[C])
        maybeSpillCollection(usingMap = false)

    // 执行所有数据的归并排序，生成一个 数据结果文件
    val partitionLengths = sorter.writePartitionedFile(blockId, tmp)

    // 写索引文件
    shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId,
partitionLengths, tmp)
```

这种普通运行机制，和 mapreduce 的shuffle 基本一致，只有一个地方不同：

```
1、mapreduce: 大小为100M 的环形缓冲区
2、spark： 如果是key-value并且有聚合操作逻辑，则使用map来进行内存的数据的保存，到一定程序在执
行溢写，如果不满足这个条件，则使用buffer来进行保存，也是到一定程度之后，执行溢写
```

mapreduce和spark的相同点：

```
1、最后每个 Task 都只会生成一个数据结果文件
3、同样，每个Task也都会为这个数据结果文件生成一个索引文件
```

5.5. Spark shuffle 源码详解

Spark-2.4.7 版本中，就只保留了 SortShuffleManager ， 他的具体实现有三种方式：

```
1、BypassMergeSortShuffleWriter
bypass机制

2、SortShuffleWriter
sortShuffle的普通通用运行机制

3、UnsafeShuffleWriter
在不需要map端聚合、partition数量小于16777216，Serializer支持relocation的情况下
```

三种不同的 shuffle，通过不同的 ShuffleHandler 来完成真正的 shuffle 处理。 ShuffleHandler 中又是通过 不同的 Writer 完成从内存到磁盘数据的输出。最大的区别在于：

```
1、在进行shuffle的时候，不同的shuffle策略，中间使用的内存结构不同：
    Map
    ArrayBuffer
2、在进行shuffle的时候，不同的shuffle策略，决定是否使用排序：
    有排序
    无排序操作
```

无论是哪种数据结构，如果要进行排序：就是使用 TimSort 算法： 稳定版本的改良的 MergeSort（把溢写到磁盘的多个 spillFile 合并成一个 spillFile， 如果要排序，最合适的算法，必然是归并）

最后的结论：

```
1、一个 Task 执行计算的是一个 RDD 中的一个分区，这个分区的数据，是以 Iterator 的形式存在的。
2、在 Spark-2.4.7 版本中，只有 SortShuffleManager，该 shuffle 的大体思路，和 MapReduce 的一致，但是稍微有些区别
    如果是key-value类型的，需要聚合的数据，则写入到map类型的数据结构中，然后根据条件执行溢写，最后执行归并排序合并成一个文件，并且生成一个索引文件
    如果是value类型的，不需要聚合的数据，则写入到buffer类型的数据结构中，然后根据条件判断是否需要执行溢写，然后生成一个分区有序的结果文件和索引文件
```

6. 本次课程总结

今天讲解的内容是 Spark Application 的提交执行和 具体的 Task 执行细节的源码剖析：

```
1、Spark Job 提交分析
2、RDD 的 DAG 构建和 Stage 切分源码详解
3、TaskScheduler 提交 Task 分析
4、Spark Task 执行源码详解
5、Spark shuffle 源码详解
```

7. 本次课程作业

使用 Spark 的 RPC 框架实现一款 C/S 架构的聊天应用程序。

要求：

1、该应用应用程序，有最基本的服务端程序和客户端程序
2、首先启动服务端，保证服务端一直运行
3、然后启动客户端，客户端向服务端注册
4、再启动其他多个客户端，待启动的客户端注册之后，和其他客户端（当然，当前客户端必须具备感知其他客户端存在的功能，然后选择通信对象）进行通信。
5、客户端关闭，其他客户端收到通知。